Abstract

Author: Andrew C. Cook

Title: Expanding Computational Thinking Across Disciplines in Higher Education

Supervising Professor: Denné N. Reed, Ph.D.

The expansion of information technology has been accompanied with growing uneasiness. As computers and code become more foundational to virtually every sector of the economy, the wide divide between the lay person and the tech guru becomes more worrisome. Accordingly, leaders have begun to call for an overhaul of the education system – we need to prepare for the economy of the future. For many high-profile advocates of this change, the solution is to push for universal programming knowledge.

Increasingly, however, researchers have seen greater potential in preparing people for a world dominated by information technology by starting with fundamental concepts, rather than the hard technical skills. In 2006, Jeannette Wing popularized the term for this approach: Computational Thinking. According to Wing, "Computational Thinking is the thought processes involved in formulating a problem and expressing its solution in a way that a computer – human or machine – can effectively carry out," (Wing 2012, p. 7). Wing and others have since worked to concretize this concept with further definitions, examples, and categories. Efforts to integrate computational thinking into education more generally have produced new courses, university programs, and literature over the results.

These efforts have not, however, been without problems, and it becomes clear from a survey of the state of the field that current conceptual understandings, categorizations, and studies of education based on computational thinking leaves much to be desired. This thesis investigates these problems and seeks to expose and partially mitigate these issues by developing a firmer understanding of computational thinking and critically analyze many of the efforts so far.

Acknowledgements

I'd like to take the opportunity to thank all of my friends and family who have patiently loved and supported me, despite my being a relatively unpleasant human being for the past many months. In particular, I appreciate the unwavering presence of my girlfriend and my parents. Seriously, thank you so much.

Table of Contents

Introduction

As a student, one of the most important things you must figure out every semester is the route to your first class. Depending on where you live and where your class is, this can be a very simple, deterministic process. For many students maybe this is trivial – the most straightforward path is one which is always the same unless something dramatic blocks their way. However, for every computer science student I have known well enough to discuss such a mundane experience, the process would only be so simple out of coincidence. Their strategies for getting to class may involve a complex decision tree that depends on traffic and weather patterns that looks something like this:

1. Walk to the elevator bank
2. If there are more people waiting for an elevator than can be expected to fit
    a. Then, turn right and walk down the stairs;
    b. Otherwise, take the elevator
3. Emerge on the ground floor and walk to the street next to campus
4. If the light is green
    a. Then, cross the street;
    b. Otherwise, if the cross walk sign is not counting down and the traffic light two lights down the street is not red
        i. Then, walk down the street to the next traffic light and cross the street when the light turns as you arrive;
        ii. Otherwise, wait for the light to change and cross the street
5. If you crossed at the second light
    a. Then, if the weather is rainy or sweltering
        i. Then, walk to and through the main building and to and through the natural science hall and emerge on the street going straight through campus
        ii. Otherwise, walk around the main building, past the turtle pond, and next to the natural science hall down to the street going straight through campus
    b. Otherwise, walk straight down to the street going straight through campus
6. Walk down the street going straight through campus to reach the building with your class in it and thereafter directly into the lecture hall.

One need not actually read through all of these instructions to get the point: computer scientists like to apply their skills to all sorts of everyday problems. They like to optimize life to a silly degree. Of course, they would argue that this isn't so silly – here you have a procedure for

getting to class that is just as deterministic (so long as you squint to ignore the fuzziness of "sweltering" and "rainy") as taking the same route every day, only this strategy is demonstrably superior in both expected time to class and quality of the walk.

My intention in making this gross and unsubstantiated generalization about how different students get to class is to illustrate how the highly technical computer science education results in many skills that are widely applicable. Mapping an efficient path to class may be a funny example, but it is both true (this was my route for one of my years in undergrad) and useful. In fact, there has been growing recognition that skills gained from computer science classes "will apply much more broadly than most of the other scientific modes of thought," (Committee for the Workshops on Computational Thinking et al. 2010). Academics have for some time begun to recognize that the high-level thought process that typically emerge as a result of training in computer science have particular value in their own right.

Unfortunately, this is not reflected in computer science education at the moment. As a professor at even Carnegie Mellon University (known particularly for its computer science program) put it, "[a]s educators, we understand that these students are learning problem-solving skills, and that they will use these skills in their careers, but many introductory CS courses focus on the programming language details, especially as the languages we use become more complex and industrial in size," (Cortina 2007). This problem seems to be quite common – introductory computer science courses often focus more on the details of particular programming languages than on the skills that computer scientists largely view as being pivotal to their job. Theoretical concepts, much more so than programming language trivia, are important for understanding computing and useful as broadly applicable skills. Cortina continues:

> After years of teaching introductory programming in a variety of languages, my observations of non-technical non-majors in such a course brought about a realization

> that these students are taking the course only to satisfy a requirement, and they are not going to program ever again in their lives. These students need to understand the power of computing and the unique problems we face in computer science that can affect their disciplines. Put another way, we would like to show them that computer science is much more than computer programming, and we only have one semester to do this. (2007)

What's particularly interesting about Cortina's charge here is that it's so focused on targeting non-majors. Obviously these concepts in computer science are helpful to computer science students as well, but that's not his primary concern. The main idea for Cortina (and, as we will see, many others) is that if students are only going to take one class in computer science, we shouldn't waste their time by teaching them the beginnings of a language they'll likely never use again. Instead, we should try to teach them the kind of skills you can use to make optimized procedures for getting to class. In other words, we should stop messing around with complex languages and dive straight into the concepts themselves.

These sentiments have coalesced into a topic area termed by Jeannette Wing's seminal work in 2006: Computational Thinking. In this paper, Wing says that

> [c]omputational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science… Having to solve a particular problem, we might ask: How difficult is it to solve? And What's the best way to solve it? Computer sciences rests on solid theoretical underpinnings to answer such questions precisely.

Wing energized the computer science community interested with expanding education and the field saw many different papers and studies pop up in the following years. Naming this term stoked interest and catalyzed new discussions about the proper place for computing in a generalized education, as well as the applicability of computer science in everyday life. One of the best things Wing did in her 2006 paper and in later work and involvement was present many concrete examples of topics in computer science that could be useful in the real world. Along the same lines as my procedure for getting to campus, she draws direct parallels for everything from

her daughter's packing her backpack for school (prefetching and caching) to coffee preparation (pipelining).

The widespread interest makes a lot of sense. Since even before Jeannette Wing rallied the academic community, computational thinking has been the focus of much interest. With the dawning of the information age, the gaps in STEM education within the United States have become increasingly apparent and concerning: How do we keep up with the rapid pace of technology? What can non-Computer Scientists do to prepare for the kind of changes we have come to expect as a matter of fact? It seems somewhat ridiculous to think that we should all become computer scientists. As with any discipline, some people clearly have a predisposition for computer science and some just don't. But it's equally clear that we miss opportunities, both with people who might have unexploited natural inclinations and with people who could benefit from some basic skills and understanding that is valuable in their interaction with technology, within their chosen disciplines, and in their lives broadly.

Accordingly, educators and policymakers have been driving a growing effort to make the importance of computing on par with english and math in curricula. The best way to address this problem long-term is to adapt primary and secondary education so as to make programming and computer science less esoteric early on in the educational process. But that kind of progress in is always difficult and slow, and in the meantime we have many students entering college who have no substantive technical experience and, as it stands now, no reasonable expectation for that to change. With the tech economy growing as fast as it is and becoming more and more influential in all industries, this problem demands attention.

Wing, Cortina, and others have begun to believe that computational thinking offers the hope of a solution. In higher education, many researchers have begun to develop programs and

courses that seek to teach computational thinking concepts. The majority of these efforts have been aimed specifically at reaching non-CS majors, in the spirit of broadening involvement and providing students with an education that actually prepares them for the information era. Two workshops have also been held to discuss the scope and nature and the pedagogical aspects of computational thinking.

Unfortunately, these efforts have spread slowly, and many problems plague the field. My impetus for investigating the topic was born from what I see as a disconnect between the optimism of the literature reinforced by positive results from many of the studies conducted so far and an adoption rate that sadly remains low. A surprising number of efforts appear to be moribund, featuring websites that haven't been updated in many years, courses that disappeared from course schedules without a trace, and unfulfilled promises for more results in later papers. Though the rate of publications remains about the same, and, encouragingly, are increasingly being adopted outside the US, efforts still fail to gain the traction that is needed to transform higher education in the way many of these academics had hoped. This thesis seeks to investigate some of the problems plaguing efforts to integrate computational thinking into general higher education curricula. It also seeks to develop frameworks for mitigating these problems, in particular by providing much-needed critical analysis of the research and efforts so far.

This investigation has identified three primary problem areas plaguing the field: poor conceptual development, inadequate topic development, and problematic research methodology. The following three chapters seek to understand and begin to address many of these problems. The broad theme to all of it is that there has been a distinct lack of critical analysis of many aspects of the topic so far. Many people have written about computational thinking, held workshops, and conducted studies, but many of the excellent ideas and information that result

remain disconnected and unexamined. Therefore, a principal aim of this thesis is to provide some degree of analysis to hopefully form a better picture of where efforts to advance computational thinking stand now and how we might build from that to produce better efforts going forward.

Chapter 1: Understanding Computational Thinking Conceptually

As it stands now, the literature on computational thinking features a broad range of disparate definitions for the term itself. As may be clear from the vagueness of the term as it has been used so far, academics tend to treat it as a catch-all to refer to mental skills that apply to computation. A workshop specifically on "the scope and nature of computational thinking" resulted in many ideas about what computational thinking is, but offered little critical analysis on the relative merits of these ideas.

This is a major problem. In order for efforts to integrate computational thinking into new curricula to make any sense, we must operate within a vigorous and consistent theoretical understanding of the concept. As Joyce Malyn-Smith argued,

> the field needs a rigorous and valid way of bringing people together and figuring out what computational thinking is. It is necessary to have consistency because not everyone understand what computational thinking is about, or they see it only through their own lens. Absent a rigorous process for defining computational thinking, efforts to promulgate computational thinking in the curriculum will lack credibility. Whatever else it may be, computational thinking in the curriculum cannot be just a bunch of examples that are placed into the curriculum at the discretion of individual teachers. (Committee for the Workshops on Computational Thinking et al. 2011)

This was opined at the last major workshop on computational thinking to have occurred – unfortunately, there has not been another in the five years since, and the literature continues to exhibit disparate understandings of the topic. As chapter 2 will discuss, developing topics and understanding them is an integral part of the process, but ultimately meaningless if it is not guided by a rigorous theoretical understanding of what is being taught. Given that such a need continues to exist, this chapter seeks to develop a definition of computational thinking as a topic area in a structured and logically justified way.

Issues with Current Definitions

A common theme of the extant definitions for computational thinking is scoping that is either extremely broad or quite restrictive. During the workshop over the scope and nature of computational thinking, for instance, Edward Fox went so far as to define computational thinking as "what humans do as they approach the world [that is, their framing, paradigm, philosophy, or language], considering processes, manipulating digital representations (and [meta] models),' and hence all humans engage in computational thinking to some extent already in their daily lives," (Committee for the Workshops on Computational Thinking et al. 2010, square brackets in original). Other advocates undermine its potential for broad appeal with their insistence on strict formality of concepts – specifically, by insisting that instruction of computational thinking involve programming languages. From the same workshop, Roy Pea, Ursula Wolz, Mitchel Resnick, and Eric Roberts all held this viewpoint, with Pea and Wolz respectively claiming that "'as soon as we think about the origins of computational thinking and computational literacies, programming has been at the heartland of the definition,[' and that] … 'programming is a language for expressing ideas. You have to learn how to read and write that language in order to be able to think in that language,'" (Committee for the Workshops on Computational Thinking et al. 2010). These sentiments are the poles of a spectrum of scoping, across which definitions fall with great variability.

Both ends of the spectrum have their advantages and disadvantages. The broad definitions are appealing because they highlight the potential of computational thinking. Many see its value as a new concept in precisely the fact that it breaks beyond the conventional notions of computer science education. A broad definition which shows how this thinking is more universal and fundamental than simple programming languages is appealing in this way in that it

absolutely escapes from the classical understanding of computational education as learning the minutiae of programming languages. Its downside is the fact that broad definitions can fail to highlight the unique aspects of a concept that make it clear what it really is. If computational thinking is just structured problem solving that everybody already uses in their lives, it doesn't actually seem that new or useful. There's nothing revolutionary to teach. The pros and cons of the the restrictive definition are inversely related: on the one hand they offer great clarity; on the other, they are uninspiring. Clearly a good definition for "computational thinking" would balance the ideas and try to best leverage the advantages of both. It would both convey rhetorically powerful meaning for the concept and suggest specific avenues of development.

Concise, High-level Definition

The one-liner that has emerged from this project's concept of computational thinking is "thinking that is constrained by and optimized for a formal system of rules that can be executed by a computer." This definition contains two operant phrases: "a computer" and "a formal system of rules." On the surface, these phrases seem pretty straight forward, but in this context they have very specific meanings which warrant some explanation to be understood. First of all, as many of the academics writing on computational thinking have pointed out explicitly, a computer need not be the electrical device we use daily to check our emails and write theses. It can be any agent that can execute instructions, including a person (Wing 2006; Committee for the Workshops on Computational Thinking et al. 2010; Denning 2003). After all, as Denning points out, "[c]omputation was [originally] taken to mean the mechanical steps followed to evaluate mathematical functions; computers were people who did computations," (Denning 2003). The second crucial part of this definition is "a formal system of rules." Broadly speaking, it's easy to understand that this means rules defining what actions are allowable, but as it's

important to distinguish computational thinking as unique from other systems, a longer discussion of the nature of this system is necessary.

Systems of Rules

To identify the sort of formal system involved in computational thinking, this section is devoted to building out the term "computational formal system," coined for this purpose. The essential nature of a computational formal system is its relation to theoretical and empirical systems and so understanding will be built around these two, disparate systems. Special effort will be put into exploring this concept, as it is reveals a great deal about the nature of computational thinking.

Theoretical systems

To begin with, we have "theoretical systems," which are formal systems that exist on a purely abstract level. Examples of these systems are different models of mathematics or logic. These systems offer a couple of key benefits. For one thing, by their nature they are entirely human constructs, meaning that it is possible and relatively easy for one person to understand and access all of the rules of the system. To put this another way, since the tools these systems provide are limited in number and complexity by their construction, it is easier to understand them deeply enough to engage in advanced creative reasoning. This allows the users of such systems to develop inferences and algorithms at a much higher level of complexity, which can result in much deeper insights. Their applicability to the empirical world is, however, much more limited as a result. They can be used to model situations, but translation from theory to reality can be problematic. The excellent insights that can result from these systems, therefore, may be trapped in the theoretical world, never improving things in reality.

Empirical Systems

To complement this, we have "empirical systems," which are simply the systems of rules that govern our empirical world. As such, everything from the laws governing matter composition and interaction to our understanding of the laws of physics to patterns of human interaction are instances of these systems. They are not understood to be formal systems, since the rules are not well-defined. Though we may often treat them kind of as though they are formal systems which we don't know in full (yet), this itself means that they are not actually formal systems. Unlike theoretical formal systems, there is no translation required for them to have real-world effects, since the reasoning is performed in a system based off the real world to begin with. The downside to this is that they are unwieldy. It is difficult to reason in these systems in ways that produce undeniably true outcomes, as is possible in theoretical systems. So even though it's really easy to make use of conclusions reached in these systems, it can be hard to reach profound conclusions in the first place.

Computational Formal Systems

Theoretical and empirical system are obviously quite different, and each have particular benefits and drawbacks. Innovation seems to occur whenever the gap between them is bridged in a way that exploits the benefits both. A hybrid system seems called for, given how limited each system is in its pure form. Theoretical reasoning has negligible value if it cannot be translated to the empirical world. Conversely, it's difficult to develop optimal solutions in the empirical world without well-defined rules. It is, therefore, incredibly valuable to bridge the gap between theory and empirics. This is the space of the formal systems of computer science.

Computational formal systems are formal (theoretical) systems for which the rules have a direct mapping to empirical laws. Computer languages and data representations so closely

represent the empirical reality of electronic signals and physical data storage (be that with physical codes, as on hard drives, or electrical charges, as with memory and solid state drives) that computer scientists are able to assume (to an extent) that the theoretical rules of the programming language translate perfectly to empirical results. In this sense, they allow computer scientists to create, as Perković termed them, "'Executable abstractions,'" (Perković et al. 2010). Unlike fully empirical systems, computational formal systems can make reliable (though, notably, not necessarily perfect) inferences and algorithms; unlike fully theoretical formal systems, they can be assumed to (though, again, do not actually) have perfect translation to the empirical world. Because computational formal systems like computer languages straddle these worlds, they can offer the primary features of each major system as reasonable guarantees, sacrificing from each only the complete guarantee.

Understanding this kind of formal system is critical to understanding computational thinking, because all computational thinking is restricted by what is allowable according to a particular computational formal system. The features of this system hint at the sort of things that are involved with computational thinking: intense creative logical reasoning and empirical test and evaluation. The scope of computational thinking can be broken down further, however.

Primary Domains of Computational Thinking

Computational thinking consists of two primary domains: problem solving (algorithm generation) and information representation (data storage). These domains are highly related to each other. Skills in both are necessary for effective computational thinking, which almost always involves extensive interplay between the two domains. It seems valuable to separate them conceptually because they are not always used together and the forms of the results are very different.

Problem solving processes

The first domain – the most widely associated with computational thinking – is a process of problem solving that blossomed in the context of programming and computer science. These steps are used to build a series of instructions (i.e. an algorithm) using a computational formal system (e.g. a programming language):

1. Design a solution conceptually (design patterns, class diagrams, etc.)
2. Verify theoretical effectiveness (runtime via big-O)
3. Implement the solution (writing the program)
4. Verify empirical effectiveness (write and run tests)
5. Identify source of errors as the are discovered (debugging)
6. Redo any of 1-4 as needed to resolve errors

When these steps are completed successfully, the result should be an unambiguous solution that a computer can execute to solve the problem at hand. Granted, the solution may not be perfect. There are limits to the creative capacity of the designer to think of every potential pitfall in execution, and even if there weren't, computers and computational formal systems themselves are imperfect and can suffer from any number of silent problems. While it's important to keep these limitations in mind, the success of programming in the face of these obstacles is proof enough of the power of this kind of problem solving process.

The significance of this domain can be seen in its prominence in understandings of computational thinking generally. One consistent theme in the workshop seeking to understand the concept was the idea of developing sets of instructions that can be executed by an agent to complete a task. Related key concepts included "effective procedure[s, which are] detailed step-by-step set[s] of instructions that can be mechanically interpreted and carried out by a specified agent," "processes and abstract phenomena that enable processes[,]" and "way[s] of formulating precise methods of doing things," (Committee for the Workshops on Computational Thinking et al. 2010). These attributes of computational thinking, put forth by various participants of the

workshop, strongly supports the idea that a primary aspect of computational thinking is a process of structured problem solving. The only inconsistency between these attributes and the definition built here is that by and large the participants of the workshop thought that structured problem solving was the single overarching domain of computational thinking, which is obviously at odds with the bimodal model presented here.

Knowledge representation

The second domain of computational thinking is conscientious knowledge representation. Computing is, at its core, information management. Programs are limited by formal systems, so computers are limited in the structure of information it can process by what the system allows. Even within a formal system, there are generally a variety of ways to represent your information. Different structures, formats, and organizational methods lend themselves to different uses. Accordingly, solving a problem within a computational formal system requires the problem solver to be deliberate in choosing how to represent the relevant information. Failure to do so can result in inefficient processing, misleading results, or missed opportunities to fill knowledge gaps in the world. Thus, an important component of computational thinking is choosing how information is represented.

This domain also is supported by the academics driving computational thinking, though not as broadly, especially within computer science. Many of the projects and discussions focused on computational thinking conceive of information representation as part of developing the procedure, which, while certainly a valid viewpoint, surrenders opportunities. In the workshop, Dor Abrahamson reportedly was concerned with computational thinking as a means of knowledge representation. The report says that he

> saw computational thinking as the use of computation-related symbol systems (semiotic systems) to articulate explicit knowledge and to objectify tacit knowledge, to manifest such knowledge in concrete computational forms, and to manage the products emerging from such intellectual efforts. He further argued that a semiotic approach had embedded within it a philosophy of the relationship between understanding and personal meaning and helps guide the construction of personal meaning for these symbols. (Committee for the Workshops on Computational Thinking et al. 2010)

This highlights a point of distinction from the other primary domain of computational thinking. By recognizing computational thinking as partly knowledge representation, Abrahamson creates the space necessary for productive conversations about data structure – including and far surpassing simple metrics of space complexity.

Synthesis

Computational thinking involves using these skills in tandem – solving problems by optimizing the interplay between a computational formal system and the empirical world and structuring information in a way that is accessible to computers. These skill sets are essential to leveraging the potential offered by computers, both by being necessary in developing programs for computers to run and by structuring information and mental thought processes in general in such ways as to make information and ideas that are generated more amenable to a computing world. For instance, it is vital for the development of the web that everyday individuals understand the limitations of publishing information to the web in text-only format and know what other options exist.

It is also important to note that this conception of computational thinking implies a certain quality in applying these skills. This is an important departure from past understandings of computational thinking; there is no "effective" and "non-effective" computational thinking. Since its constituent skills involve, for instance, solution analysis and verification for proper information structure, thinking that does not investigate and optimize its effectiveness is not

computational thinking. This is important because it addresses one of the problems created by broad definitions: if everybody is engaging in computational thinking all the time (since we all engage in structured problem solving and information representation to some extent), then it's more like a natural phenomenon than a skill which can be taught. With the concept of computational thinking advanced here, however, the optimizing aspect of it clearly differentiates it from processes that do not involve vigorous verification.

Conclusion

In the end, the only thing that makes a definition good is when people choose to use it. This aim with this chapter was not to create a new standard and argue for its superiority, but rather to posit an understanding of computational thinking for *this* project. As pointed out, this definition has many advantages, which hopefully others will recognize and agree with, and it has been a useful guide for the project as a whole. The bigger impetus behind it, however, is to present an example of a well-explained and well-justified definition. Any definition used as part of a project should, for instance, should point out why it is useful and serves the goals of computational thinking. It should clearly differentiate computational thinking from similar ideas, such as logic, mathematics, the scientific method, and broad problem solving strategies so that is becomes easy to say what is and is not computational thinking.

Chapter 2: Understanding Topics

As was briefly mention at the beginning of the previous topic, advocates for computational thinking have a tendency to develop lists of topics in order to understand what the term means and to guide the development of curricula. To understand why, we should look back to when Jeannette Wing started the trend in her seminal paper. When she wrote out terms lists of buzzwords in computer science along with accompanying non-technical examples, she stimulated academics' imaginations. In discussing these concepts with very plain language and offering some justification for their importance outside of computer science, Wing was highlighting the big idea behind computational thinking. She was saying that these ideas need not be so scary – we can teach them to anybody. Topics highlight the abstract nature of computer science. They are concepts that computer scientists recognize and understand instantly on an high level, without much regard for programming details. Stuff like "concurrency," "heuristics," and "modularity" are very deeply understood in the field of computer science, but they also exist on a much higher conceptual level than programming language knowledge, which indicates their potential as educational topics. They seem quite easily explained without relying on computer code, as demonstrated by the many parallels proffered by Wing.

At first glance, this seems to forget Malyn-Smith's assertion, quoted at the beginning of chapter 1, that "computational thinking in the curriculum cannot be just a bunch of examples that are placed into the curriculum at the discretion of individual teachers" (Committee for the Workshops on Computational Thinking et al. 2011). Remember, however, that her criticism was not against developing lists of topics per se, but rather against doing so without connection to a deeper framework or consideration for how it fits into the field at large. The work done in chapter 1 was meant to address precisely this concern and as such, the understanding of topics

here is informed by the definitional framework created previously. Topics and examples are grounded, especially in an understanding of computational formal systems.

So, these topics are important and useful for teaching people computing in that they are well understood and nicely encapsulate the things we want people to understand. They are, in fact, vital to understanding computational thinking as a topic area for education programs, because these are the topics that will show up in syllabi. Most studies that develop courses enumerate topics to some extent, and build lectures and class assignments around them. In making them, educators have the freedom of course to decide which topic areas are sufficiently important, what we should call them, and whether and how they ought to be grouped, so there is no strict list of topics. As Robert Constable pointed out in the workshop for the scope and nature of computational thinking,

> rather than a finite set of skills and thought processes, computational thinking is an open-ended and growing list of concepts that reflects the dynamic nature of technology and human learning, and that combines elements of all the descriptions of computational thinking outlined [in the workshop] such as "automating intellectual processes" and "studying information processes," among others. (Committee for the Workshops on Computational Thinking et al. 2010)

Even Wing herself stressed the breadth of these topics, with her long and non-comprehensive lists of computational thinking topics and examples.

Still, Constable, Wing, and others all agree on the importance of listing these topics. This chapter seeks to follow this strategy and develop a deeper understanding of computational thinking by presenting a list of topics and subtopics. This list of topics is the synthesis of several different lists found in the literature. Each topic is followed by a concrete example, with special effort made to highlight non-technical applications or technical applications for non-technical endeavors. The purpose for this exercise is twofold. First, while many academics develop topics for their programs and classes, they often are not backed up with thorough explanation, which

assumes deep prior computer science knowledge. As each topic here will be accompanied with an explanation and non-technical example, hopefully this list will be more accessible to educators who may not have extensive computing experience. Second, as this list presumes to be neither complete nor authoritative, it means to serve as a sort of template for future efforts to identify and explain topics in computational thinking.

Algorithms

One of the most common topic areas was simply termed "algorithms" (Wing 2006; Dierbach et al. 2011; Cortina 2007; Perković et al. 2010; Denning 2003). At its most basic level, an algorithm is a series of instructions that perform some specific action. Perković[1] defined it as "a process that starts from an initial state containing the algorithm and input data, and goes through a sequence of intermediate states until a final, goal state is reached," (Perković et al. 2010). This topic seems like it must be too broad to be useful, but it is, nonetheless, a topic which can and must be taught in order to understand computing. True, an algorithm is just a list of instructions, but making that list and understanding that list become quite advanced matters when you're working in a computational formal system. Your list of instructions must comply with all of the restrictions imposed by the rules of the system you are using. You must understand what your algorithm is designed to do, or – in Perković's words – what the beginning and end states should be and how we go from one to the other. This level of precision does not come naturally.

A great example of an algorithm is a series of chess moves developed by a player to capture an enemy piece. Chess players operate in a computational formal system. Tjhe rules are

[1] Note that Perković's categories are exactly from Denning's "Great Principles of Computing," but with different definitions for the terms. I use many of these principles/topics in this chapter, taking variously from both Perković and Denning's definitions for them.

well defined as the moves available to the chess pieces and the boundaries of the board and each individual square. This series of moves could be as small as single move or as long as five moves which force a checkmate. The initial state is the board configuration before any steps are carried out, the final state is board configuration without the targeted piece, and the intermediary steps are the board configurations after each player's move. So, anybody who has ever played chess has engaged in a process of conceptualizing an algorithm (thinking about the next sequence of moves) and executing that algorithm (actually making the moves).

Search Strategies

Algorithms come in all sorts of flavors, and there are classic algorithms that are heavily studied and theoretically understood. An example of this came up multiple times as a topic in its own right: search strategies (Committee for the Workshops on Computational Thinking et al. 2010; Wing 2006). These are algorithms that are looking for a particular item in a group of items (usually presumed to be of the same type). Some of the most basic search strategies include linear search and binary search, though there are more complicated modifications of these. Let's explain directly with an example.

First there is linear search. Suppose you have a deck of cards, and you are looking to find the jack of spades as fast as possible. A linear search strategy would be for you to start at the bottom of the deck and look at each card successively until you find the card. In the abstract this can be defined as a list of instructions:

1. Check the card on the bottom of the deck. If it is the card in question, you're done, otherwise;
2. Remove the bottom card such that a new card is on the bottom
3. Repeat

It should be pretty easy to see that given a correct initial state (a card to search for and a deck of cards that contains that card), this series of instructions will always find the requested card.

Second we have binary search. Suppose now that you have a dictionary with 1000 pages and are trying to find the page containing a word you want to look up. A binary search strategy would have you flip to page 500 and check if your word is between the two words on the top of the page. If it is, you're done! If not, you know your word is either on pages 1 – 499 or pages 501 – 1000, depending on whether your word comes before or after the words at the top of the page. You then repeat this process – choose your middle page (either 250 or 750) and check again. Every time you do this process, you'll be looking at smaller and smaller sections of the dictionary – 1000 pages goes to ~500 pages goes to ~250 pages and so on – until you have eliminated all of the dictionary from consideration except for one page, at which point, you are done! The word, of course, still needs to be found on the page, but this can be done using a similar strategy.

These search strategies are fundamental algorithms of computer science, yet can clearly be taught without venturing into the world of code whatsoever. The real world examples might seem too cumbersome in their formality to actually be useful in real life, but this formality can give you some advantages in different situations and inform your actions when you are searching for something.

Coordination

Another topic area that is even terminologically familiar to lay people is "coordination," (Perković et al. 2010; Denning 2003). Coordination is an important topic because computer systems often involve many different independent computers or processes depending on one another to fulfill a task. As Denning puts it very simply, coordination is "[e]ffectively using many autonomous computers," (Denning 2003). In computer science, this problem area most heavily depends on how often and for how long computers need to wait around for each other.

Solving these sorts of problems is actually a rather advanced topic in computer science, though it's pretty easy to see parallels in normal life. To illustrate, the sections below will demonstrate coordination in the subtopics of communication and concurrency using the example of students working in a group project.

Communications

According to Perković, one of the ways computers coordinate with each other is through communication (Perković et al. 2010). Also identified as topic by Wing (2012), communication plays a vital role in modern computing by facilitating information exchange. The topic could, in fact, be reduced entirely to making sure that computers send the correct data to each other at the best time and in the best way. Computer systems have neither perfectly reliable nor instantaneous communication systems so it is therefore important to choose appropriate methods of communication.

Students in a group project understand these decisions naturally in deciding, for instance, whether the group needs to meet in person or if exchanging a few emails will suffice. Group members may weigh the benefits in group understanding of the in-person meeting against the relative ease and speed of emails. If consensus and understanding is very important for an assignment, it'd probably be best to make the effort to meet in person; if members live far apart and travel time would be prohibitive, the emails might be heavily preferred. Programmers similarly must balance questions of reliability and latency (how much time communication requires).

Concurrency

Concurrency (often also called parallel processing or parallelization) was one of the most commonly mentioned topic in the literature (Committee for the Workshops on Computational Thinking et al. 2010; Wing 2006; Cortina 2007; Kafura and Tatar 2011), which is interesting considering how advanced the topic is in computer science. The idea is pretty simple: split up the work of a program between multiple computers or multiple processors on a single computer. It's obviously better to divide work between different computers and processors as much as possible – things go twice as quickly if you can have two different computers working on them – but there are risks associated with doing this. If a computer can come in and change a data source another computer is working on, the second computer can never be totally sure that the data remains the same without taking additional measures to specifically check for this. Concurrency, then, focuses on dividing the work in a way that avoids these problems, or mitigates the risks if they are unavoidable.

The parallel in the group project is pretty direct. To maximize group efficiency, group members need to split the work in a sensible way. Let's say they are making a powerpoint presentation. The most inefficient strategy is that which uses no concurrency – all group members work on the same slide at the same time. Slightly better but still not optimal would be for half of the group members to work on the visual design of the slides (relatively easy) with the other half working on the content (relatively hard). The best strategy depends on details about the presentation, but could be, for instance, having each team member build one or more slides on their own, to be combined and standardized later. The important point here is how we evaluate the different strategies. Optimal strategies keep all the group members working as much as

possible and therefore seek to minimize dependence between group members on each other. This lesson is also key to concurrency in programming.

Design

Design is another topic area identified by researchers (Denning 2003; Perković et al. 2010) which functions more as an umbrella category rather than as an independent skill set. Broadly speaking, "[d]esign is the organization… of a system, process, object, etc." (Perković et al. 2010). As opposed to algorithms, Design is about structure definition rather than a list of instruction (though algorithms themselves have structure). It can be thought of as setting up the conditions in which the algorithms will run.

Data Organization

This topic, which is concerned with the structure and storage methods of information, came up in a few different names in the literature (Cortina 2007; Denning 2003; Perković et al. 2010). Data can be formatted in vast array of different structures, which have different advantages and disadvantages. The organization of the data in the computer (e.g. different file types) restricts what sort of programs can access it, and informs the use of the data. In programming, different theoretical structures control how data is accessed and processed.

For lay people, this is a particularly important topic because it's quite common to make decisions about data organization for computing. For instance, it generally is much better to make a balance sheet with an Excel spreadsheet rather than with a Word document – though it is entirely possible with both – because spreadsheets have many properties which make maintaining the balance sheet less laborious and which allow useful features such as graphing. On the flip side, it really makes no sense to write an essay in a spreadsheet.

Broadly speaking, these are the kinds of question programmers often must answer in the course of their work – what's the best organization for this data, as informed by the ways I expect to use it?

Modularization

Modularization is a subtly different topic identified in the literature (Committee for the Workshops on Computational Thinking et al. 2010; Wing 2006). Rather than thinking about the structure and organization of the data that is to go through a program, modularization considers the structure and organization of the program itself. Programs are often divided into different parts for different purposes, because clearly differentiating task definitions makes the program much easier to understand for computer scientists. If programmers need to change some code, it's easier if tasks are clearly distinguished from one another to minimize the risk of making changes with unintended consequences. For another thing, if a program is actually made of many smaller programs, future programmers can make use of some of the individual parts that might apply in completely different contexts.

An excellent direct example of this topic in the real world is the historical example of interchangeable parts. This production technique allows people to easily exchange components of their product, since the specifications of the parts and product are well defined. So, for instance, we know that one particular size of bolt goes on a particular part of a bicycle, because the bolt and the screw where the bolt will go were designed specifically to screw together and hold fast when tightened. It's much easier to identify problems (say, a stripped bolt) and apply a solution (screwing on a new bolt) because the bike isn't a single specialized machine, but rather a collection of many different commonly available parts. There is also the additional advantage that the bolt need not only be used on the bike; it can be used on different types of bikes, or

perhaps on a lawnmower. The distinctness of the bolt from other parts on the bike and ubiquity of the bolt in everyday life reflect the nature of its modularity.

Good programmers learn to develop with modularity in mind, because of the ease of use and usefulness it provides. Similarly, modularity can help everyday people develop solutions to problems that leverage the same sort of advantages, whether in bike repair or business structure.

Conclusion

As mentioned previously, this list is by no means exhaustive. The topics listed here, in fact, would be quite inadequate for a semester-long university course intended to give students a broad understanding of computational thinking. Many of them do, however, represent some of the most important concepts in the field. This section is important, because it demonstrates the sort of things that educators would like students to learn in a computational thinking course. Delving deeply into these topics can change student's perspectives on everyday activities which never seemed consequential as formal ideas before, but which actually have extensive theoretical support and understanding in the computer science community. They may learn strategies for solving problems or interacting with the world generally that they never thought of before. These topics may also spark an interest for computer science that a traditional introductory course with its concentration on teaching language details can miss. If nothing else, they give students a much better understanding of what programming actually involves.

Hopefully the usefulness of these topics in stimulating educational development are also clear. These clearly defined and widely applicable skills offer great flexibility in terms of how they are taught and give educators valuable resources while still granting great latitude in forming a course.

Chapter 3: Review and Critical Analysis of Literature

Since the inception of the concept, computational thinking has sought to transform education standards and curricula to make computing more accessible. Accordingly, this topic has been investigated in a number of educational contexts, including primary, secondary, and higher education. Within higher education, the focus of this thesis, there is further stratification. Efforts range from fostering useful familiarization in non-technical majors to improving introductory and fundamentals classes for CS majors. It is, of course, great that so many researchers think that the topic area is promising enough to spend the time, money, and energy necessary to create these curricula and publish their results. If nothing else, it is clear that a large number of academics (primarily from computer science) see a great deal of education potential in computational thinking and are willing to investigate.

The state of the field, however, suffers from disorganization and shortcomings in the literature. For one thing, many of the studies are not that informative. For a number of reasons, the studies often are not very scientific, meaning that it is difficult to confidently draw conclusions from the study or replicate the process, and basically impossible to justify efforts in the field with data about results. Furthermore, despite two major conferences, the results of the studies remain largely disconnected and (much like with the definitions in chapter 1) lack substantive critical analysis – studies often cite each other, but seem to do so only to compare methodologies. Seldom are results addressed or compared.

For these reasons a thorough literature review seems to be called for, in which we look at what has been done so far and draw a picture of how the topic area stands now. The studies under examination result from computational thinking courses at Baylor University (Booth et al. 2013), Virginia Tech University (Kafura and Tatar 2011; Kafura, Bart, and Chowdhury 2015),

Union College (Barr 2012), University of Massachusetts Lowell (Martin et al. 2009), DePaul University (Perković et al. 2010; Settle 2011), Towson University (Dierbach et al. 2011), Carnegie Mellon University (Cortina 2007), Purdue University (Hambrusch et al. 2009), Tuskegee University (Qin 2009), the University of Texas at San Antonio (Yuen and Robbins 2014), the National University of Defense Technology in China (Li and Wang 2012), and an Israeli middle school using similar strategies but with arguably better methodologies (Meerbaum-Salant, Armoni, and Ben-Ari 2013). Nine of these studies were for courses intended for non-technical non-CS-majors (Booth et al. 2013; Kafura, Bart, and Chowdhury 2015; Martin et al. 2009; Perković et al. 2010; Settle 2011; Cortina 2007; Dierbach et al. 2011; Meerbaum-Salant, Armoni, and Ben-Ari 2013; Li and Wang 2012), three were for STEM majors (Hambrusch et al. 2009; Qin 2009; Yuen and Robbins 2014), and two were made for exclusively computer science majors (Kafura and Tatar 2011; Barr 2012).

These studies span a wide swath of teaching strategies, scientific methodologies, and student populations. Unsurprisingly, their content varies quite widely. Nonetheless, consistent themes emerged, which should be informative to studies moving forward.

Common Methodology Issues

Extant studies on computational thinking unfortunately leave much to be desired as far as vigorous scientific research goes. This poses a number of difficulties for researchers moving forward. For one thing, despite a healthy number of computational thinking courses having been developed, researchers have little hard data to inform their efforts. For another thing, good scientific data plays a critical role in garnering continuing support for studies and curricula changes from the scientific community and school administrators.

In many ways, the heart of the problem may stem from the the studies' largely being conference papers, and are therefore not subjected to the same standards as journal papers. Some of the studies have basic errors in methodology which presumably a more thorough review process would expose, so there is something to be said for the context of these studies – one shouldn't expect conference papers to achieve the same level of rigor. This itself is a problem, though. After spending the time and effort necessary to develop these courses and, in many cases, methods for their evaluation, it's a shame that the results stop at the conference level, especially when they so often include as-yet-unfulfilled promises for further inquiry and results. This likely results from difficult obstacles common to research, such as scarcity of funding and administrative politics, as well as some more specific to computational thinking, such as computer science education rarely being the principle's primary research area.

There are, nonetheless, some more specific problems which ,are common themes throughout the literature. Hopefully an analysis of them could reduce the probability of their occurring in the future.

Enrollment

Many studies had problems boosting enrollment without creating a requirement for the course, with the big exception being Cortina's 2007 study at CMU. This is a huge barrier for computational thinking programs. Poor enrollment hurts results and is a failure in its own right, given that that entire point of computational thinking is to making computing topics more accessible. It would seem that the stigma against anything "computer" is strong enough to keep students away from even courses designed to help bridge this educational gap. In other words, it appears that students are not themselves eager to learn computing skills at this time (with, it seems, the unsurprising exception of CMU).

This problem is not an easy one to solve. It's difficult to fight prevailing attitudes about computing through persuasion, no matter how important to their future lives it may be. Perhaps an appropriate response would be to make computational thinking a general requirement, as was the case with Barr's 2012 study at Union college, and as many of the other studies seemed to be planning, given their courses' being meant for general education. As a corollary, it is also important to note that poor gender ratios plagued these studies as well. It's well documented how this is a problem for computer science generally, with female enrollment in the field calculated to be .14 in 2004 (Wine, Janson, and Wheeless 2011), so one would hope that courses designed to engage students broadly would especially seek to engage women, given that the population is underrepresented and therefore laden with untapped potential.

Data Quality

An additional problem that is clear from these studies is the fact that the data are often not of high quality. For one thing, an inordinately high number of studies rely on either purely anecdotal information or anecdotal interpretations of survey data without giving the results themselves (Barr 2012; Dierbach et al. 2011; Martin et al. 2009; Qin 2009), with one study actually giving no results whatsoever (Perković et al. 2010). Many of the studies that do have some data are not much better. Some rely exclusively on end-of-course surveys to determine the success of the course (Kafura, Bart, and Chowdhury 2015; Cortina 2007; Kafura and Tatar 2011; Li and Wang 2012). Precious few give evaluations over the course of the class (Booth et al. 2013; Meerbaum-Salant, Armoni, and Ben-Ari 2013; Settle 2011; Hambrusch et al. 2009). Fewer still use well established evaluation metrics (Kafura, Bart, and Chowdhury 2015; Booth et al. 2013). Only one was longitudinal in any respect, and even then, the only things that were considered over the course of time were enrollment and grades (Cortina 2007). This last point is

particularly important, because as it stands there is really no evidence arguing that computational thinking is actually worth integrating into curricula at all.

If these efforts are to continue and grow, it is vital that studies collect better data on a longer time scale. Metrics exist, such as the Computational Thinking Problem Solving Inventory, developed as a part of the study at Baylor University (Booth et al. 2013), and the Computer Anxiety Rating Scale, also used by the study at Baylor University and developed by Heinssen Jr., Glass, and Knight (Heinssen, Glass, and Knight 1987). It also shouldn't be too tall an order for researchers to conduct longitudinal studies, especially at CMU and Virginia Tech, where courses in computational thinking have been offered at least since 2007 and 2010, respectively.

Negative Results

This leads to another problem with the research. Of the studies under consideration, only two are in the position to have these good quality longitudinal studies because computational thinking courses have an alarming tendency to disappear. An investigation of the American universities' course catalogues reveals that the vast majority of these efforts have been, at best, put on hold or, at worst, abandoned entirely, since the courses are in most cases no longer offered. The worst part about this situation, though, is that the cause of this is pure guesswork. As alluded to previously, many of these efforts frustratingly die off after promising to publish more results in the future, seemingly encouraged by the work so far. Rarely are difficulties in the classes discussed seriously – much less so the structural obstacles of the sort that might prompt the abandonment of such a program.

This is a massive problem. While we can speculate on the nature of these obstacles, we really need the researchers themselves to report on what happens by identifying problem areas either initially or ex post facto. A mitigating strategy could be an effort designed specifically to

investigate and report on the causes of these failures, perhaps while simultaneously looking towards the successful studies to try to identify some of the features that led to their success. To a large extent, the only data available at the moment are the structures and limited results of these studies, so it's vital to have information about the fate of the programs themselves.

Common Findings

The poor state of the field should not, however, compel us to throw out the results from these studies. They're what the field has now. Even anecdotal findings can help guide future studies, though they do not provide the firm foundation that would be optimal. Below are some common themes seen in the findings of the studies.

Teaching Success

One thing that seem clear across a number of studies is that the classes that are developed can successfully teach students many of the topics that are fundamental to computer science without relying heavily on coding knowledge. This is seen in self-reporting by students at the end of a class (Booth et al. 2013; Kafura and Tatar 2011), personal evaluation by professors (Booth et al. 2013; Dierbach et al. 2011), and student assessment (Booth et al. 2013; Meerbaum-Salant, Armoni, and Ben-Ari 2013; Li and Wang 2012). While this finding isn't particularly groundbreaking, it does show that computing topics can be taught to students in a variety of disciplines and in a variety of contexts. This supports one of the fundamental assumptions behind the interest in computational thinking: computing topics are much more widely accessible than current curricula reflect.

More Favorable Perspectives

The most significant result by far is that many studies have shown that courses in computational thinking lead to interest in further computer science courses (Kafura and Tatar 2011; Kafura, Bart, and Chowdhury 2015; Dierbach et al. 2011; Cortina 2007; Hambrusch et al. 2009), decreased computer anxiety (Booth et al. 2013), increased confidence in pursuing computer science in the future (Kafura and Tatar 2011; Cortina 2007; Hambrusch et al. 2009), and widespread recognition of the usefulness of computer science (Kafura, Bart, and Chowdhury 2015; Dierbach et al. 2011; Li and Wang 2012). In fact, a positive or improved outlook on technology and computer science was the most consistent theme emerging from the studies under review here. This is very encouraging, because the primary aim of curricula based on computational thinking is to demystify technology for students. The fact that students come away from these courses more positive about computing strongly indicates that computational thinking does indeed fulfill this critical goal.

The results are even more encouraging on a granular level. For instance, Cortina reports that "the increase in enrollment in this new introductory course is not based on the chance to obtain a higher grade. The average grade in both classes[, which were taught by the same instructor in the same time frame,] was a B, and fewer students earned A's in the computation course, yet the enrollment continues to increase," (Cortina 2007). The same study reported that 85% of students indicated they would recommend the course to their friends. The ubiquity of this finding should encourage any educators who are considering developing a new computational thinking course or program.

Difficulties

Despite this, some studies provide a word of warning for educators to be careful in making sure that students are exposed to topics in the right way, using the proper tools, and with consideration for their level of technological proclivity. Some studies reported that students at times felt confused (Dierbach et al. 2011; Yuen and Robbins 2014) or that the topics were difficult to grasp (Li and Wang 2012). Others reported that the teachers at times felt inadequately prepared in the topics or tools being used in the course (Meerbaum-Salant, Armoni, and Ben-Ari 2013; Qin 2009). Given that perceived difficulty is one of the main reasons that students choose against pursuing computer science (Gundermann and Frantz 2008), future efforts should be sure to guard against course content outstripping student understanding.

At the same time, it is equally important to make sure that instructors are adequately prepared to teach the course effectively. Two studies in particular mention difficulties encountered by the instructors, especially with regards to the technologies being used themselves (Qin 2009; Meerbaum-Salant, Armoni, and Ben-Ari 2013), while two others discussed strategies for preparing faculty with the topic using workshops (Booth et al. 2013; Dierbach et al. 2011). Of course, the preparation required depends heavily on the structure of the study. Most courses in this literature were designed and taught by computer science professors, meaning that they were already intimately familiar with both the topics and the technology. Some sort of training is necessary when instructors or contributors come from other disciplines.

Learning Tools

One of the most common ideas in computational thinking was that introductory computing instruction should move away from being focused on learning fundamentals of languages, as is often the case now. Of course, given the nature of computing, tools are important

for teaching process flow and data storage and manipulation. The choice of tool, then, depended primarily on the context of instruction.

Basically all studies involving students that were new to computing used very simple block-based programming languages to some extent, with many reporting that taking such a strategy was helpful for students (Kafura, Bart, and Chowdhury 2015; Meerbaum-Salant, Armoni, and Ben-Ari 2013; Li and Wang 2012). The tools used in these studies included Blockly, Scratch, Raptor, Alice, and NetLogo, with the last being the only language that was reported as having questionable effectiveness for students. As indicated above, however, it is important to ensure that educators are familiar with the tool being used and that the environment is stable. Technical issues are easily preventable obstacles to student learning and effective lab time.

As far as professional languages go, python was used in one study for non-CS majors, python and MATLAB for STEM majors, and a variety of advanced technologies in the courses for on computer science students. This makes some intuitive sense. Python has the simplest syntax of the major languages and, being a scripting language, requires essentially no confusing structure, and studies support its effectiveness as an introductory language. The study that used it with non-technical majors, however, did highlight the importance of using it as a late topic in the course. The courses for STEM majors likely used MATLAB because data visualization and analysis is an integral part of their fields, and as such they likely have a proclivity for the technology anyway. Finally, computer science courses, of course, would naturally involve a wide variety of technologies, as it is important in the field to be exposed to a variety of technologies starting early.

Visualization

A common feature of almost all studies was the focus on visualization and/or easily relatable data, with four distinct studies reporting student responses affirming the importance of these things in their findings (Cortina 2007; Kafura and Tatar 2011; Yuen and Robbins 2014). The basis for this focus is the belief that the main challenge in teaching people computing knowledge is the abstract and foreign nature of programming constructs. For most people, their education up to college treats computing as a niche subject area on par with say, theater tech: nice to learn if you have an interest in it, but likely not pivotal to your future. The subject area, therefore, is unfamiliar to many, and could benefit from as many connections to the real world as possible to provide grounding. Real-world data and/or problems do this by demonstrating to students that the work they aren't doing isn't theoretical. Visualizations are representations of data and processes that students are better equipped to understand. As far as visual programming goes, block-based languages have the added benefit of letting students largely ignore questions of syntax and instead spend more time and effort on understanding the core concepts themselves.

Collaboration

The final common feature of these studies was the fact that the courses in almost all of these studies involved team-based project development or learning, with three distinct studies reporting that the collaboration was indeed effective (Kafura, Bart, and Chowdhury 2015; Settle 2011; Yuen and Robbins 2014). This sort of learning environment is quite common in computer science. Instructors reported that putting students in groups often reduced the strain on individual students, because they have a resource in each other that is more available, approachable, and (in many cases) understandable than either the instructor(s) or any materials that were provided. One study in particular reported that, before collaborative working was allowed in the course, the

students were “uncomfortable during one-to-one interaction with a computer during computer laboratory exercises” and that they had “problems with spelling, and many errors… due to mis-typed commands and parameters,” (Qin 2009). All of these challenges were reportedly greatly reduced when students were paired for the lab exercises.

This all strongly supports the already widely held belief that collaborative learning is particularly valuable in computing contexts. With a partner, simple errors that aren’t at the heart of the concept in question are often resolved much more quickly, preventing time wasted by stumped students. Furthermore, group work is valuable for many students because it helps teach and reinforce skills of professional interaction that are central to any career, especially in computing careers, in fact, despite the stereotype. As Qin also acknowledged, collaborative learning and working comes with the increased challenge of effectively assessing individual performance, but it seems clear that the benefits of collaboration justify the additional burden.

Conclusion

The state of the studies on computational thinking so far leaves a lot to be desired. Efforts to date just have not translated into good results very often. If computational thinking is to be taken seriously as an educational area, the academic community needs to produce consistently better quality results. This likely depends on many different things. Perhaps we need to get better about selling the idea to administrators in order to get better institutional support for these new curricula. Likely it would also help to do better at branching out of computer science departments when developing these problems.,which would likely involve more talks, workshops, etc. to engage and train non-CS faculty. Money is likely another issue for researchers, suggesting the need for more nation-wide support from the academic community (though, of course, every field has this complaint).

One avenue for future work that could be very useful is actually contacting some of the researchers who have run studies. Given that most studies have failed without warning, the professors leading those efforts likely have some insight to offer for things that future projects should avoid. Additionally, given that Virginia Tech University and Carnegie Mellon University (and, to a lesser extent, Union College) are examples of programs that have managed to succeed somehow, future efforts could seek to learn from professors like Cortina and Kafura in order to either figure out better what worked so well for them so future studies may emulate their programs' successes, or work with these schools to conduct more comprehensive analyses of these programs than currently exist.

Though there is a lot to complain about for the research efforts undertaken so far, there is reason to be optimistic. On the whole, results were very positive. Despite the poor data and plainly incomplete understandings of programs' lifespans, consistent themes across the studies keep the hope alive for the potential of computational thinking. Hopefully investigators act on this hope to better turn it into a reality.

Concluding Remarks

People shouldn't be anxious about computing. That is the underlying assumption driving this thesis. Though they certainly exist, we don't need studies to tell us that we struggle to get people – especially women and minorities – interested in computer science. Given the way education is structured currently, it's quite possible for people to glide through their entire education without learning what computing is really about or why it might be interesting to them. Computational thinking seeks to address that, but beyond that, it seeks to give students skills that will help them moving forward, regardless of their field or industry.

I began this thesis with an illustration of how my computer science education shapes my everyday life – an optimized process for getting to class. It's a silly example – people aren't aching to root out every little inefficiency in their life – but much more meaningful ones exist. Computational thinking can lead to unique insights in other fields. In philosophy I found my familiarity with logic quite helpful when trying to follow the complicated reasoning of Kant. In my humanities classes, I find that my tendency to try to break my argument or mess up my literature interpretations helps me understand topics in the class and makes my responses to them much better. When unexpected problems arise when planning activities with a group of friends, I find that my ability to quickly reconstruct a problem in such a way as to make it easier to solve can often resolve issues quickly. Computational thinking is in many ways a different way of looking at the world, and one which can give people deeper insight about just about any aspect of life. It's unfortunate that research has so far failed to quantify these advantages, given that computer scientists know quite well how understanding computational thinking concepts can broadly enrich one's life.

Admittedly, all of this is rather grand for what in practice turns out to be a single-semester course. I don't think anybody is under any delusions with regards to that, though. Proper education is always a long and difficult process, and there's nothing about computational thinking that makes it prone to fantastical world-changing successes. If computational thinking courses led to moderate upticks in computer science enrollment, academics who support the idea would likely be pretty happy.

All of this hinges on engagement, though. This thesis is an interesting project because it focuses on an educational idea that is specifically concerned with engaging diverse groups of people, and yet for the most part it remains the side hobby of computer science academics. Ironically, very few of the studies examined in chapter 3 involved principle non-CS faculty. Definitions, topics, and studies won't even matter if the field doesn't practice what it preaches and engage with other disciplines. That's why this is so appropriate for a Plan II thesis. Plan II – the "interdisciplinary honors program," as my one-liner has evolved into over the years – has no required courses that deal with computing. The scant non-required classes that even touch technology at all usually do so only in the context of some other field, for instance, with the anthropology of technology, rather than with technology itself. The reason for this is non-engagement. If our field became better at exporting knowledge – perhaps through something like computational thinking courses, workshops, and programs – we wouldn't find it so often relegated as ancillary/esoteric knowledge best suited for optional electives when curricula are being crafted.

Computational thinking offers a great deal of hope in these endeavors. Many see it as the key to bridging disciplines and de-mystifying a great deal of the technological world in which we all now live. By necessity, understanding the complex rules of programming languages may

always be challenging, even to the extent that it acts as a barrier, but this doesn't mean that the field itself is off limits to all but those who dare to brave Java and C++. It is clearly possible for all kinds of people to not only understand but also gain value for concepts that are fundamental to computer science. Hopefully future efforts will build on all the work that has already been done so that computational thinking can become a staple of a liberal education.

Bibliography

Barr, Valerie. 2012. “Create Two, Three, Many Courses: An Experiment in Contextualized Introductory Computer Science.” *J. Comput. Sci. Coll.* 27 (6). USA: Consortium for Computing Sciences in Colleges: 19–25.

Booth, William A., Greg Hamerly, David Sturgill, Ivy Hamerly, and Todd Buras. 2013. “Computational Thinking: Building a Model Curriculum.” *ACET Journal of Computer Education and Research* 8 (’). http://acet.ecs.baylor.edu/journal/ACETJournal_Vol8/Computational%20Thinking.pdf.

Committee for the Workshops on Computational Thinking, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences, and National Research Council. 2010. *Report of a Workshop on The Scope and Nature of Computational Thinking*. National Academies Press.

———. 2011. *Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. National Academies Press.

Cortina, Thomas J. 2007. “An Introduction to Computer Science for Non-Majors Using Principles of Computation.” In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, 39:218–22. SIGCSE ’07. New York, NY, USA: ACM.

Denning, Peter J. 2003. “Great Principles of Computing.” *Communications of the ACM* 46 (11). New York, NY, USA: ACM: 15–20.

Dierbach, Charles, Harry Hochheiser, Samuel Collins, Gerald Jerome, Christopher Ariza, Tina Kelleher, William Kleinsasser, Josh Dehlinger, and Siddharth Kaza. 2011. “A Model for Piloting Pathways for Computational Thinking in a General Education Curriculum.” In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, 257–62. SIGCSE ’11. New York, NY, USA: ACM.

Gundermann, Dawn M., and Case K. Frantz. 2008. “Evaluation of the Emerging Scholars Program in Computer Science (ESP-CS) 2005-2007.” KD Evaluation Consultants. http://www.pltlcs.org/reports/PLTL_NSF_2008_Report.pdf.

Hambrusch, Susanne, Christoph Hoffmann, John T. Korb, Mark Haugan, and Antony L. Hosking. 2009. “A Multidisciplinary Approach Towards Computational Thinking for Science Majors∗ .” In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, Volume 41:Pages 183–87. SIGCSE ’09. ACM.

Heinssen, Robert K., Jr., Carol R. Glass, and Luanne A. Knight. 1987. “Assessing Computer Anxiety: Development and Validation of the Computer Anxiety Rating Scale.” *Computers in Human Behavior* 3 (1): 49–59.

Kafura, Dennis, Austin Cory Bart, and Bushra Chowdhury. 2015. “Design and Preliminary Results From a Computational Thinking Course.” In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 63–68. ITiCSE

’15. New York, NY, USA: ACM.

Kafura, Dennis, and Deborah Tatar. 2011. “Initial Experience with a Computational Thinking Course for Computer Science Students.” In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, 251–56. SIGCSE ’11. New York, NY, USA: ACM.

Li, Tun, and Ting Wang. 2012. “A Unified Approach to Teach Computational Thinking for First Year Non–CS Majors in an Introductory Course.” *IERI Procedia* 2 (January): 498–503.

Martin, Fred, Gena Greher, Jesse Heines, James Jeffers, Hyun Ju Kim, Sarah Kuhn, Karen Roehr, Nancy Selleck, Linda Silka, and Holly Yanco. 2009. “Joining Computing and the Arts at a Mid-Size University.” *Journal of Computing Sciences in Colleges* 24 (6). Consortium for Computing Sciences in Colleges: 87–94.

Meerbaum-Salant, Orni, Michal Armoni, and Mordechai (moti) Ben-Ari. 2013. “Learning Computer Science Concepts with Scratch.” *Computer Science Education* 23 (3). Taylor & Francis: 239–64.

Perković, Ljubomir, Amber Settle, Sungsoon Hwang, and Joshua Jones. 2010. “A Framework for Computational Thinking Across the Curriculum.” In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, 123–27. ITiCSE ’10. New York, NY, USA: ACM.

Qin, Hong. 2009. “Teaching Computational Thinking Through Bioinformatics to Biology Students.” In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, 188–91. SIGCSE ’09. New York, NY, USA: ACM.

Settle, Amber. 2011. “Computational Thinking in a Game Design Course.” In *Proceedings of the 2011 Conference on Information Technology Education*, 61–66. SIGITE ’11. New York, NY, USA: ACM.

Wine, Jennifer, Natasha Janson, and Sarah Wheeless. 2011. “2004/09 Beginning Postsecondary Students Longitudinal Study (BPS:04/09) Full-Scale Methodology Report.” NCES. http://nces.ed.gov/pubs2012/2012246.pdf.

Wing, Jeannette M. 2006. “Computational Thinking.” *Communications of the ACM* 49 (3). ACM: 33–35.

Yuen, Timothy T., and Kay A. Robbins. 2014. “A Qualitative Study of Students’ Computational Thinking Skills in a Data-Driven Computing Class.” *Trans. Comput. Educ.* 14 (4). New York, NY, USA: ACM: 27:1–27:19.

Andrew C. Cook was born and raised in Lubbock, TX and lived in the same house until he moved to Austin to attend the University of Texas. There he majored Plan II Honors and Computer Science, taking honors computer science courses throughout undergraduate as a part of the Turing Scholars Honors program. During his summers, he worked at Cisco Systems, Bloomberg LLP, and Google as an intern all across the country. For the fall of his fourth year, he studied at Universität Salzburg in Salzburg, Austria. During all other semesters, he worked a lot on his many honors courses and made many dear friends who were somehow working even harder. After graduation in December 2016, he will be going back to work as a full-time employee at Bloomberg LLP in New York City.